Automatic Algorithm Selection in Computational Software Using Machine Learning

Matthew C. Simpson  
NC State University  
Raleigh, NC 27695  
Email: mcsimps2@ncsu.edu

Qing Yi  
University of Colorado  
Colorado Springs, CO 80918  
Email: qyi@uccs.edu

Jugal Kalita  
University of Colorado  
Colorado Springs, CO 80918  
Email: jkalita@uccs.edu

Abstract—Computational software programs, such as Maple and Mathematica, heavily rely on superfunctions and meta-algorithms to select the optimal algorithm for a given task. These meta-algorithms may require intensive mathematical proof to formulate, incur large computational overhead, or fail to consistently select the best algorithm. Machine learning demonstrates a promising alternative for automatic algorithm selection by easing the design process and overhead while also attaining high accuracy in selection. In a case study on the resultant superfunction, a trained neural network is able to select the best algorithm out of the four available 86% of the time in Maple and 78% of the time in Mathematica. When used as a replacement for pre-existing meta-algorithms, the neural network brings about a 68% runtime improvement in Maple and 49% improvement in Mathematica. Random forests, k-nearest neighbors, and both linear and RBF kernel SVMs are also compared to the neural network model, the latter of which offers the best performance out of the tested machine learning methods.

I. INTRODUCTION

The algorithm selection problem was first formalized by Rice [1] and is stated as follows:

Given the space of all problems $\mathcal{P}$, along with an algorithm space $\mathcal{A}$ which contains all known algorithms to solve the problems in $\mathcal{P}$, determine a selection mapping $S : \mathcal{P} \to \mathcal{A}$ that maximizes performance for each problem $x \in \mathcal{P}$.

That is, if we measure performance in $\mathbb{R}^n$, where there are $n$ dimensions of performance to take into consideration (e.g., runtime, memory usage, error), and if we define a performance measure $p : \mathcal{A} \times \mathcal{P} \to \mathbb{R}^n$ that maps an algorithm applied to a problem instance to its performance in $\mathbb{R}^n$, we wish to find the aforementioned selection mapping $S$ such that for any $a \in \mathcal{P}$, it holds that $||p(S(x), x)|| \geq ||p(a, x)||$ for each $a \in \mathcal{A}$.

In essence, $S$ finds the algorithm that offers the best performance for any problem instance.

A robust solution to the algorithm selection problem is especially important for NP-Hard problems, where the algorithm runtimes can be highly variable based on the inputs to the problem. Mathematical and scientific computational software, such as Matlab, Mathematica, Maple, Sage, and NumPy, have largely resorted to employing superfunctions and meta-algorithms as a solution to this problem. Superfunctions are methods that encapsulate function calls to more specific methods to compute what the user desires. For example, the superfunction $\text{dsolve}$ can be called by a Maple user to solve a system of ordinary differential equations, but it does so by, in turn, calling more specific subroutines that are available to solve such systems, such as the Taylor series method or the Rosenbrock method. To determine which specific method to call, these superfunctions use a meta-algorithm, which is responsible for making an educated choice as to which algorithm will perform the best, usually with regards to runtime.

Algorithm selection ultimately relies on properties of the inputs. Some inputs simply just won’t work with certain algorithms, and other algorithms are able to provide performance enhancements and error reductions if the input is well conditioned. The latter opens up an opportunity for machine learning to be the ultimate selector of which algorithm to employ. Meta-algorithms rely on preconceived notions and rules of thumb about which algorithm should be the best in a given situation, rather than on statistical data about what has been the best approach. Designing meta-algorithms can be extremely complicated, especially to take in as many features of the inputs as possible. It is very likely some features are missed or will have to be ignored for the sake of computation. The difficulty of design is escalated when multiple aspects of the output are important, such as when both runtime and error reduction need to be taken into account instead of just one or the other. In addition, the complicated design of meta-algorithms can add large overhead to what may seem to be a simple task. Sometimes, this overhead is more than that of the ultimate algorithm selected [2].

Machine learning can be used in practice to replace meta-algorithms by acting as a classifier to analyze important features of the input and then classify the input into which algorithm would be appropriate for it. This approach allows the program to make use of a wider feature set to make more precise decisions, compared to the smaller feature set the designers of meta-algorithms are usually forced to focus on. This added precision brings about a higher accuracy in the proportion of times the best algorithm is selected. Machine learning also avoids the need to create rules of thumb based on each feature and eases the design process needed to create an algorithm selection tool. Such an approach can be applied to any superfunction with performance dependence on inputs. For each superfunction, the important input features that affect performance need to be extracted to train the model, a general process which does not require significant tailoring to any
specific problem as meta-algorithm generally do. Additionally, the problem of adding newly implemented algorithms to the pool of ones to select from becomes trivial; the machine learning model need only be retrained to account for these new algorithms, which takes a matter of seconds, compared to a total rewrite and overhaul design of meta-algorithm code. Lastly, machine learning can take into account multiple facets of runtime when choosing an algorithm, such as runtime, memory usage, and error, whereas meta-algorithms typically have to focus on only one of these aspects.

This paper aims to make headway by using machine learning as a tool for automatic algorithm selection (AAS) in computational software. The resultant superfunction, available in most mathematical programs, is used as a case study for this goal. In addition, Maple and Mathematica are used to evaluate the results of machine learning and compare them to their corresponding meta-algorithm implementations for the resultant. The main technical contributions of this paper are:

- Developing a general approach and necessary formulations for using machine learning to automate the selection of algorithms in computational software,
- Demonstrating, through the resultant superfunction, the success of machine learning as a tool for AAS, and
- Empirically comparing the performance of machine learning models to that of Maple’s and Mathematica’s meta-algorithms for the resultant.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 introduces the mathematical background of the resultant and the algorithms used for its computation. Maple’s meta-algorithm for choosing amongst these algorithms is also discussed, as well as the features and datasets used for training multiple machine learning models. Section 4 evaluates the accuracy of different machine learning approaches and compares these results to those of Maple and Mathematica, the main ideas of which are again summarized in Section 5.

II. RELATED WORK

Meta-algorithms are not the only approach to the algorithm selection problem. The algorithm portfolio paradigm [3] was an early attempt at a solution, which, in effect, selects a portion of all available algorithms and runs them in parallel until one of them finishes. The parallel computation causes a large resulting overhead, but nevertheless, this method shines for certain problem classes with heavy-tailed runtime distributions. In these cases, the algorithm portfolio paradigm is more advantageous than running a single algorithm [4]. Dynamic algorithm portfolios [5] provide a step forward from traditional algorithm portfolios by running a set of algorithms in parallel, but then iteratively updating the priority of each process by how well each is performing. The biggest problem here, however, is developing and implementing a measure in each algorithm that allows one to judge the current progress made. More importantly, this measure needs to be designed such that it allows for a fair comparison between different algorithms.

For many end users in scientific and mathematical software, these parallel computing approaches are not feasible given their large overhead. Although these approaches minimize runtime, no regard is given to the resource management aspect of performance. Because of this, meta-algorithms have been selected as the preferred tool in computational software.

The idea of using statistical and machine learning techniques for the selection of algorithms is not unheard of. Brewer [6][7] brought about the idea of using regression for performance predictions by using linear fitting to predict the runtime of different implementations of multiprocessor libraries on unseen architectures. In the field of meta-learning, Bradzil et al. [8] applied this technique using an Instance-Based Learning approach to select appropriate learning algorithms for different sets of problems. Doan and Kalita [9], [10], [11] used regression models to predict algorithm performance for different data mining techniques. Similarly, the study in this paper continues to build upon this idea in the context of computational software.

III. THE RESULTANT CASE STUDY

To demonstrate the overarching hypothesis that machine learning can be a better alternative to meta-algorithms for use in computational software, a popular superfunction was chosen as a case study, namely the resultant. The resultant is a fundamental tool used in computer algebra, algebraic geometry, algebraic cryptography and elimination theory, and it finds higher level use in algorithms for integration, solving systems of nonlinear equations, computing the discriminant of two polynomials, analyzing greatest common divisors, and so forth.

A. The Resultant

Given two polynomials \( a, b \in \mathbb{F}[x_1, x_2, \ldots, x_k] \), where \( \mathbb{F} \) denotes an integral domain, of degree \( n \) and \( m \) respectively, they can be written with respect to the variable \( x_l, 1 \leq l \leq k \), as \( a(x_l) = a_n x_l^n + \ldots + a_1 x_l + a_0 \) and \( b(x_l) = b_m x_l^m + \ldots + b_1 x_l + b_0 \), where \( a_i, b_i \in \mathbb{F}[x_1, \ldots, x_l-1, x_l+1, \ldots, x_k] \). The resultant function taken with respect to \( x_l \) is defined as the determinant of Sylvester’s matrix [12][13].

\[
S_{x_l}(a, b) = \begin{vmatrix}
    a_n & a_{n-1} & \cdots & a_0 & 0 & \cdots & 0 \\
    0 & a_n & a_{n-1} & \cdots & a_0 & \cdots & \vdots \\
    \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
    0 & \cdots & 0 & a_n & a_{n-1} & \cdots & a_0 \\
    b_m & b_{m-1} & \cdots & b_0 & 0 & \cdots & 0 \\
    0 & b_m & b_{m-1} & \cdots & b_0 & \cdots & \vdots \\
    \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
    0 & \cdots & 0 & b_m & b_{m-1} & \cdots & b_0
\end{vmatrix}
\]

The resultant turns out to be of interest because computing the determinant of the Sylvester’s matrix is equivalent to the following.
\[ \text{Res}_x(a, b) = \det(S_{x1}(a, b)) = a_n^m b_n^m \prod_{r \in \mathbb{F}[x_1, \ldots, x_k]} (r_a - r_b) \]

For monic polynomials, this is the product of the pairwise difference in the roots of each polynomial. This becomes especially important for handling the roots of polynomials in an algebraic fashion, that is, without actually having to explicitly compute their value. Thus, in computation, the roots are never solved for, because that is a problem in and of itself and defeats the purpose of handling the roots algebraically.

Although more exist, four algorithms are generally used to compute the resultant: Sylvester’s matrix, Bezout’s matrix [14], Collin’s modular resultant [15], and Brown’s subresultant pseudo-remainder sequences [16]. All four of these algorithms are available in Mathematica and Maple.

The computation of the resultant via Bezout’s matrix is quite similar to doing so with Sylvester’s matrix, given above. Bezout’s matrix is defined as 

\[ B(a, b) = (b_{ij})_{i,j=1,\ldots,\max\{n,m\}} \]

with elements 

\[ b_{ij} = \sum_{k=1}^{\min\{n+1-r_j, m+1-r_i\}} a_{ij-k} b_{i-k} - a_i b_{j-k}. \]

The determinant of this matrix is a multiple of the resultant of \( a \) and \( b \).

Collin’s modular resultant algorithm derives from the idea that taking the determinant of a matrix, such as Sylvester’s matrix, only requires addition and multiplication operations. Thus, it makes sense to take advantage a homomorphism \( \phi : X \to Y \), where it holds \( \phi(x_1 + x_2) = \phi(x_1) + \phi(x_2) \) and \( \phi(x_1 x_2) = \phi(x_1) \phi(x_2) \) for any \( x_1, x_2 \in X \). In this case, it also holds then for a matrix \( Q = (q_{ij}) \) with elements \( q_{ij} \in X \) that \( \phi(\det(Q)) = \det(\phi(Q)) \). Collin originally applied this result to the domain of integers \( \mathbb{Z} \) using the mapping \( \phi : \mathbb{Z} \to \mathbb{Z}_p \), where \( p \) is a prime integer, to take advantage of the fact that if \( |\det(Q)| < \frac{p}{2} \), then \( \phi(\det(Q)) = \det(Q) \), and thus, by the previous homomorphic equality, \( \det(\phi(Q)) = \det(Q) \). Typically, to make use of smaller primes, several primes and homomorphic mappings are used to do the reduced calculations, the results of which can then be recombined using the Chinese remainder theorem.

Brown’s subresultant algorithm is a generalization of the Euclidean algorithm [17] for computing the greatest common divisor of two integers or polynomials. For polynomials \( a, b \in \mathbb{F}[x_1, x_2, \ldots, x_k] \), assume, without loss of generality, that \( \deg(a) \geq \deg(b) \). When the division algorithm holds, \( a \) can be written as \( a = b \cdot q + r \), where \( q, r \in \mathbb{F}[x_1, x_2, \ldots, x_k] \) and are called the quotient and remainder, respectively. If the division algorithm does not hold, pseudo-remainderers are used instead by introducing a constant multiplier to \( a \). The Euclidean algorithm makes use of the fact that \( \gcd(a, b) = \gcd(b, r) \), so we can recursively apply this reduction until the remainder is 0. The subresultant algorithm uses a similar equality but applied to subresultants, which come from submatrices of Sylvester’s matrix. These are denoted \( \text{Res}_{x_j}(a, b) \) as the \( j \)-th order subresultant, which effectively eliminates \( j \) rows and columns from Sylvester’s matrix. The following equality then holds:

\[ \text{Res}_{x_j}^j(a, b) = (-1)^{(n-j)(m-j)} b_{m}^{j-\deg(r)} \text{Res}_{x_{j+1}}^j(b, r). \]

This allows us to iteratively simplify the problem, just like the Euclidean algorithm.

B. Meta-algorithms

Mathematica’s meta-algorithm to select among the four available algorithms is hidden, but Maple’s is available in the documentation on the resultant superfunction and by using the showstat command to view the source code. For univariate and bivariate polynomials with rational coefficients (including integral coefficients, even though the two are stored in memory differently), Maple uses modular methods for high degree polynomials, whereas the subresultant algorithm is used for those with lower degrees. In all other cases, Bezout’s matrix is used. It’s worth noting that, even though Maple offers Sylvester’s matrix as an option, it is never considered by the meta-algorithm. Without a doubt, this is because, in most cases, taking the determinant of a smaller matrix, such as Bezout’s matrix, is faster than taking the determinant of a larger matrix, such as Sylvester’s matrix. However, this assumption ignores important factors, such as the computation of the elements in Bezout’s matrix, whether any elements are zero, whether floating point numbers are being used, the kernel of the program, and so on. These factors can regularly make Sylvester’s matrix a more viable method; in fact, for randomly sampled polynomials, the proportion of times Sylvester’s matrix outperforms all other algorithms is close to the same proportion of times Bezout’s matrix does so too.

C. Machine Learning Formulations

Machine learning can be applied to the algorithm selection problem by using classification to categorize a pair of input polynomials into which algorithm will work best for them. This classification relies only on a set of attributes or features that are taken from a quick analysis of the two polynomials \( a \) and \( b \) passed in as arguments during the function call \( \text{resultant}(a, b, x_i) \), the resultant of \( a \) and \( b \) with respect to the variable \( x_i \). Feature engineering can be completed by analyzing the source code for implemented algorithms or by having a general understanding of each algorithm and then determining the fundamental features of the inputs on which the runtime complexity depends.

The basic features on which to classify are generally easy to spot. For example, it is clear to see that all four available algorithms to compute the resultant depend on the degrees of the input polynomials, although in different manners; the dimensions of Bezout’s and Sylvester’s matrices directly depend on the degrees of the polynomials, and the coefficient bounds of Collin’s modular method changes with the degrees of the polynomials, as well as the number of iterations that occur in the subresultant algorithm.
creates a weight for each feature by analyzing the
takes a random feature vector from the training data and
from an initial size of 30. The ReliefF algorithm iteratively
rowed down using the ReliefF algorithm [18] to 18 attributes
modular method’s runtime may change significantly.
not change at all, as might happen with the fact that Sylvester’s
factorially. It’s also possible that an algorithm’s runtime may
cost to take the determinant of Sylvester’s matrix scales
regards to changes in any given feature. For example, the
recognition tool and consisted of a single hidden layer with
hits, which are the closest feature vectors under the L1 norm
SVMs with linear and RBF kernels were trained on the given
Table I.
A. Setup
To train and test the machine learning models, all com-
putation was done on a computer with an Xeon E5-2420
CPU, Matrox G200eR2 video controller, and 16 GB of RAM,
running 64 bit CentOS 6.8 with an installation of Maple 2015.1
and Mathematica 10.0.2.

<table>
<thead>
<tr>
<th>No.</th>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True/False: $a, b \in \mathbb{Z}[x_1, \ldots, x_k]$</td>
<td>Whether or not polynomials have all integer coefficients</td>
</tr>
<tr>
<td>2</td>
<td>True/False: $a, b \in \mathbb{Q}[x_1, \ldots, x_k]$</td>
<td>Whether or not polynomials have all rational coefficients</td>
</tr>
<tr>
<td>3</td>
<td>True/False: $a$ or $b$ has floating point coefficients</td>
<td>Whether or not computations will require floating point precision</td>
</tr>
<tr>
<td>4</td>
<td>Number of indeterminants</td>
<td>Number of variables in both polynomials combined</td>
</tr>
<tr>
<td>5</td>
<td>$deg(a)$</td>
<td>Degree of $a$ with respect to $x_1$</td>
</tr>
<tr>
<td>6</td>
<td>$deg(b)$</td>
<td>Degree of $b$ with respect to $x_1$</td>
</tr>
<tr>
<td>7</td>
<td>Number of terms in $a$</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>8</td>
<td>Number of terms in $b$</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>9</td>
<td>Sparsity rating of $a$</td>
<td>(Number of nonzero coefficients of $a)/(deg(a) + 1)$</td>
</tr>
<tr>
<td>10</td>
<td>Sparsity rating of $b$</td>
<td>(Number of nonzero coefficients of $b)/(deg(b) + 1)$</td>
</tr>
<tr>
<td>11</td>
<td>Number of algebraic coefficients in $a$</td>
<td>The number of terms that consist of a variable besides $x_1$</td>
</tr>
<tr>
<td>12</td>
<td>Number of algebraic coefficients in $b$</td>
<td>What portion of coefficients are purely algebraic in $a$</td>
</tr>
<tr>
<td>13</td>
<td>Proportion of algebraic coefficients in $a$</td>
<td>What portion of coefficients are purely algebraic in $b$</td>
</tr>
<tr>
<td>14</td>
<td>Proportion of algebraic coefficients in $b$</td>
<td>The numeric coefficient of the highest order term in $a$</td>
</tr>
<tr>
<td>15</td>
<td>$lcoeff(a)$</td>
<td>The numeric coefficient of the highest order term in $b$</td>
</tr>
<tr>
<td>16</td>
<td>$lcoeff(b)$</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>17</td>
<td>Numeric coefficient with smallest magnitude out of $a$ and $b$</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>18</td>
<td>Numeric coefficient with largest magnitude out of $a$ and $b$</td>
<td>Self explanatory</td>
</tr>
</tbody>
</table>

It is usually the case that algorithms scale differently with
regards to changes in any given feature. For example, the
cost to take the determinant of Sylvester’s matrix scales
polynomially with the degrees of the inputs, whereas the
coefficient bound in the modular resultant algorithm scales
factorially. It’s also possible that an algorithm’s runtime may
change with a given feature, whereas another algorithm’s may
not change at all, as might happen with the fact that Sylvester’s
matrix could care less about the polynomial ring, whereas
the modular method’s runtime may change significantly.

After enumerating many of these features, the list was nar-
rowed down using the ReliefF algorithm [18] to 18 attributes
from an initial size of 30. The ReliefF algorithm iteratively
takes a random feature vector from the training data and
creates a weight for each feature by analyzing the $k$ nearest-
hits, which are the closest feature vectors under the L1 norm
with the same classification, and the $k$ nearest-misses from
each different class, which are the closest feature vectors with
a different classification. The full list of attributes used for the
resultant case study, along with their descriptions, is given in
Table 1.

As for labeling, there were four target classes for each of
the four available algorithms. Classification into one of these
categories means the resultant algorithm corresponding to that
class gives the best performance in terms of runtime.

The dataset consisted of 18,346 randomly generated poly-
nomials, which were randomly paired into 9,173 inputs, as the
resultant function takes two polynomials as a single input.
Since the learning was supervised, the output class for each
input in the dataset was obtained by running each of theour algorithms, which are already implemented in Maple and
Mathematica, and selecting the resultant algorithm that gave
the least runtime over an average of thirty runs.

Neural networks, random forests, k-nearest neighbors and
SVMs with linear and RBF kernels were trained on the given
data. The neural network was built with Matlab’s pattern
recognition tool and consisted of a single hidden layer with
10 sigmoid hidden neurons and a softmax output layer. These
formed a feed-forward network that was trained with scaled
conjugate gradient backpropagation. The data for the neural
network was split into 70% training, 15% validation, and 15%
testing.

The random forest model was built based off the model in
[19] with 50 random trees that, at each node, split on a random
selection of five features. Forests with more trees did not
offer significant gains in accuracy compared to the associated
cost in runtime, and splitting among less than five features
caused a loss in accuracy. For the k-nearest neighbors model,
classification was done based solely on the nearest neighbor,
since using multiple neighbors caused the accuracy to drop off
steeply. A linear search with the Euclidean distance norm was
used to find the nearest neighbor. Both the random forest and
the k-nearest neighbors model were trained with 10-fold cross
validation. Lastly, the SVM model was trained with both a
linear and a RBF kernel through libsvm’s interface. For these
three methods, data was divided into 80% training and 20%
testing.

IV. Experimental Results

A. Setup

To train and test the machine learning models, all com-
putation was done on a computer with an Xeon E5-2420
CPU, Matrox G200eR2 video controller, and 16 GB of RAM,
running 64 bit CentOS 6.8 with an installation of Maple 2015.1
and Mathematica 10.0.2.

The performance of machine learning as an automatic
algorithm selection tool was compared to Maple’s and Math-
ematica’s meta-algorithms. More tools for comparison exist,
such as Sage, but these tend to default to simply computing
the determinant of Sylvester’s matrix instead of choosing
amongst the different algorithms available. To test the benefits
of machine learning, neural networks, random forests, k-
nearest neighbors, and SVMs were each trained, and the two
most accurate of which, namely neural networks and random
forests, were used to replace Maple’s and Mathematica’s meta-algorithm to select the ultimate resultant algorithm used. These two were then run and timed over several thousand problem inputs. Note, however, that the output class for a pair of inputs was obtained by running all algorithms implemented by the given program and taking the one that gave the least runtime. Since Maple and Mathematica have different kernels which are optimized for different operations, the best algorithm when run under Maple may not be the same as the best algorithm when run under Mathematica for a given input. Thus, it would not be wise to use the machine learning model trained on the best algorithms determined by Maple’s runtimes as a replacement for Mathematica’s meta-algorithm. Keeping with this paradigm, two datasets were generated, one of which based its output classes on the best algorithm when run under Maple and the other when run under Mathematica.

B. Accuracy

The accuracies, or proportions of the time when the algorithm with the least runtime out of all four available algorithms was correctly chosen, for each machine learning approach used are given in Table II. For training on the data generated under the best algorithm choices determined by runtimes in Maple, neural networks outperformed the other machine learning models, correctly selecting the best algorithm 86.63% of the time on the testing data (86.17% and 85.54% for training and validation respectively). Random forests managed to attain 80.71% accuracy, but k-nearest neighbors, RBF kernel SVMs, and linear kernel SVMs all lagged behind at 77.46%, 76.68%, and 76.63% accuracy, respectively.

However, these accuracies indicate that the associated machine learning models are vast improvements over the use of meta-algorithms. Maple’s meta-algorithm selects the best algorithm out of all four available only 58% of the time. (The accuracy of the meta-algorithm can be determined by looking at the source code or the userinfo output to see when a specific algorithm has been called.) However, as previously discussed, considering the meta-algorithm only ever opts for three out of the four available (namely, it ignores Sylvester’s matrix), the accuracy in choosing the best out of three is 72%. Unfortunately, in close to 15% of cases, Sylvester’s matrix proves to be the best choice, so although Maple can select amongst the modular, subresultant, and Bezout algorithms with a 72% accuracy, this choice is still not the best 15% of the time. Nevertheless, in either accuracy measurement, all tested machine learning models outperformed Maple’s meta-algorithm.

Machine learning’s performance dropped significantly when tested against the data generated under Mathematica. Neural networks only attained a 78.24% accuracy, and random forests came close with a 75.97% accuracy. K-nearest neighbors, RBF SVMs, and linear SVMs all underperformed at 69.03%, 70%, and 67% accuracy respectively. Unfortunately, Mathematica hides the implementation of its meta-algorithm and provides no way to see which algorithm was ultimately selected by its meta-algorithm, so the accuracy of Mathematica’s choices remain unknown.

C. Time Speedup

Although there are obvious differences in accuracies between machine learning models and meta-algorithms, these differences are compounded by significantly faster runtimes when machine learning models are used to replace meta-algorithms during the selection of algorithms stage. When applied to a random sample of several thousand inputs, Maple was able to compute the resultant of all inputs in 37,783 seconds with its original meta-algorithm, whereas using the neural network as the selection tool when the resultant superfunction was called yielded a total runtime of only 12,097 seconds, a 68% decrease in runtime. Similarly, in Mathematica, the neural network brought about a 49% decrease in runtime. Random forests, on the other hand, only brought a 46% decrease in runtime to the same sample in Maple and a 37% decrease in Mathematica. Since neural networks have such a better runtime improvement compared to random forests, even though their accuracies only differ by close to 6%, it appears as though when the neural network made an incorrect algorithm choice, the decision it did make was not as bad as when the same situation occurred for random forests. In the case of random forests, the incorrectly chosen algorithm was typically also not the second best algorithm choice. Nevertheless, the high accuracies still resulted in significant performance gains. Since running these tests takes large amounts of server time and resources, the runtime results are limited to just neural networks and random forests and are summarized in Table III.

It serves to note that there is evidence that Mathematica’s meta-algorithm uses pre-processing to speed up the computation of the ultimately selected algorithms, whereas a direct call to a specific resultant algorithm does not use this pre-processing. This claim comes from the fact that, in occasional instances in the dataset, a call to Mathematica’s meta-algorithm yields a faster runtime than single-handedly calling any one of the four algorithms directly. Yet, despite this disadvantage, using machine learning as an automatic algorithm selection tool still manages to outperform Mathematica’s meta-algorithm by a significant factor.

V. CONCLUSION

The 68% and 49% runtime improvements attained when replacing Maple’s and Mathematica’s resultant meta-algorithms

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy on Maple data</th>
<th>Accuracy on Mathematica data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neural Networks</td>
<td>86.63%</td>
<td>78.24%</td>
</tr>
<tr>
<td>Random Forests</td>
<td>80.71%</td>
<td>75.97%</td>
</tr>
<tr>
<td>KNN</td>
<td>77.46%</td>
<td>69.03%</td>
</tr>
<tr>
<td>RBF SVM</td>
<td>76.68%</td>
<td>70%</td>
</tr>
<tr>
<td>Linear SVM</td>
<td>76.63%</td>
<td>67%</td>
</tr>
</tbody>
</table>
with a neural network, respectively, support the hypothesis that machine learning provides a better alternative for automatic algorithm selection in computational software. The overall methodology emphasizes the fact that this process is not only straightforward but highly effective. Since most resultant applications require the computation of the resultant dozens or hundreds of times, the runtime improvement would be quite noticeable. In fact, even for a single pair of input polynomials of which to compute the resultant, it is not uncommon for the possible runtimes to range from several seconds to a couple minutes depending on which algorithm is selected. The high accuracy achieved by machine learning avoids such excessive runtimes.

What remains in the works is to expand this approach to more case studies and to develop a multi-objective machine learning model to maximize different facets of performance. This would enable an algorithm selection model that can select the best algorithm based on not only runtime performance, but also based on memory constraints, least error, and so on. Such is particularly useful for approximation algorithms, wherein the user requires a reasonable runtime and a high accuracy in computation. For instance, evolutionary multi-objective optimization algorithms, such as NSGA-II [20] and SPEA-II [21], may be easily used to perform such computation, although the time required to do so may not be acceptable but needs to be empirically determined. All in all, the results in this paper have built a foothold for further exploration into the benefits of machine learning as a tool for automatic algorithm selection, especially in computational software.

ACKNOWLEDGEMENT

This work was funded under NSF grant 1359275. The authors would also like to thank Nathan Harmon for his valued input on the project.

<table>
<thead>
<tr>
<th>Model</th>
<th>Runtime Improvement in Maple</th>
<th>Runtime Improvement in Mathematica</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neural Networks</td>
<td>68%</td>
<td>49%</td>
</tr>
<tr>
<td>Random Forests</td>
<td>46%</td>
<td>37%</td>
</tr>
</tbody>
</table>

### REFERENCES